Kauno technologijos universitetas

Informatikos fakultetas

Inžinerinis projektas

P170B328 Lygiagretusis programavimas

Projekto autorius
Gustas Klevinskas

Akademinė grupė
IFF-8/7

Vadovas
Dominykas Barisas

Kaunas, 2020

Turinys

Užduotis

Pasirinkau išlygiagretinti skaitinių metodų ir algoritmų antro laboratorinio optimizavimo uždavinį.

Duotos n $(3 \leq n)$ taškų koordinatės $(-10 \leq x \leq 10,\ -10 \leq y \leq 10)$. Srityje $(-10 \leq x \leq 10, -10 \leq y \leq 10)$ reikia pridėti papildomų m $(3 \leq m)$ taškų taip, kad jų atstumai nuo visų kitų taškų (įskaitant ir papildomus) būtų kuo artimesni vidutiniam atstumui, o atstumas nuo koordinačių pradžios būtų kuo artimesnis nurodytai reikšmei S $(1 \leq S)$.

Šį optimizavimo uždavinį pasirinkau spręsti kvazi-gradientiniu metodu. Mano aprašyta tikslo funkcija susideda iš trijų įverčių: atstumo tarp stacionarių ir pridėtų taškų, atstumo tarp pridėtų taškų tarpusavy ir pridėtų taškų atstumo nuo centro įskaičiuojant pasirinktą dydį S.

Algoritmas baigia darbą, kai žingsnis gradiento kryptimi tampa mažesnis nei 10^{-3}. Kiekvienoje iteracijoje apskaičiuojamas naujas artinys ir nauja tikslo funkcijos reikšmė. Jei ji mažesnė, žingsnį padidiname 5%, jei didesnė, kitaip tariant tikslas buvo peršoktas, grįžtame į praeitą artinį ir žingsnis sumažinamas 5 kartus.

Taikyto algoritmo parametrai:

- išvestinės žingsnis – 0.001;
- pradinis žingsnis gradiento kryptimi – 0.1.

Skaitinių metodų ir algoritmų laboratorinis buvo rašytas naudojant Python, tad lygiagretinimui pasirinkau naudoti `multiprocessing.Pool` klasę.

Programos pagrindinių dalių aprašymas

Duotų ir papildomų taškų *x* ir *y* koordinatės yra saugojamos masyvuose, kurie perduodami funkcijoms atlikti skaičiavimus.

Pagrindinių funkcijų aprašymai pateikti sąraše žemiau. Be jų yra dar kelios pagalbinės funkcijos, kurios parodo koordinačių sistemoje taškus, nuskaito failą su taškų koordinatėmis, apskaičiuoja tikslo funkcijos reikalavimų dalis.

Funkcija `solve()`

Tai yra pagrindinė funkcija, kurioje gaunamas uždavinio sprendimo rezultatas. Čia nuskaitomi iš duomenų failų duoti taškai ir pradiniai artiniai pasirinktiems taškams.

Funkcijos `gradient()` ir `gradient_mp()`

Šiose funkcijose apskaičiuojamas gradientas pagal duotus taškų masyvus. `gradient_mp()` funkcija skiriasi tuo, kad joje gradientas apskaičiuojamas lygiagrečiai. Kadangi gradiento skaičiavimui reikia rasti dalinę išvestinę pagal visas pasirenkamų taškų koordinates, ši vieta geriausiai išlygiagretinama.

Funkcija `psi()`

Ši funkcija apskaičiuoja tikslo funkcijos reikšmę pagal jai perduotus duotus ir parinktus taškus.

Funkcija `psi_prime()`

Joje apskaičiuojama dalinė išvestinė pagal per funkcijos argumentus paduotus pridėtų taškų indeksus.

Instaliavimo ir paleidimo instrukcija

Programos paleidimui reikia Python3 interpretatoriaus ir šių bibliotekų:

- `matplotlib` – apskaičiuoto rezultato parodymui;
- `numpy` – pagalbinės matematinės funkcijos.

Jas galima instaliuoti naudojant pip:

```
pip install matplotlib numpy
```

Programa paleidžiama su komanda:

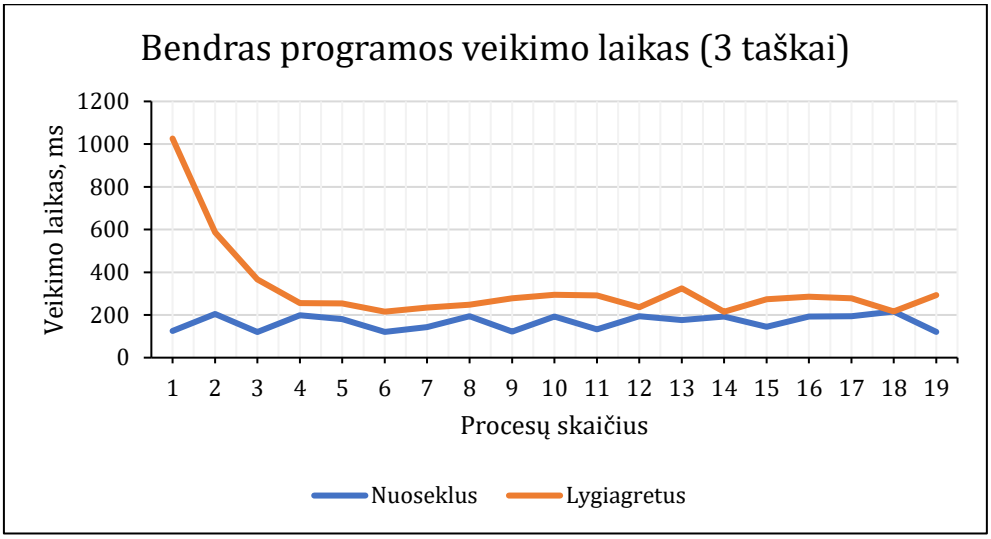
```
python gradient.py
```

Rezultatai

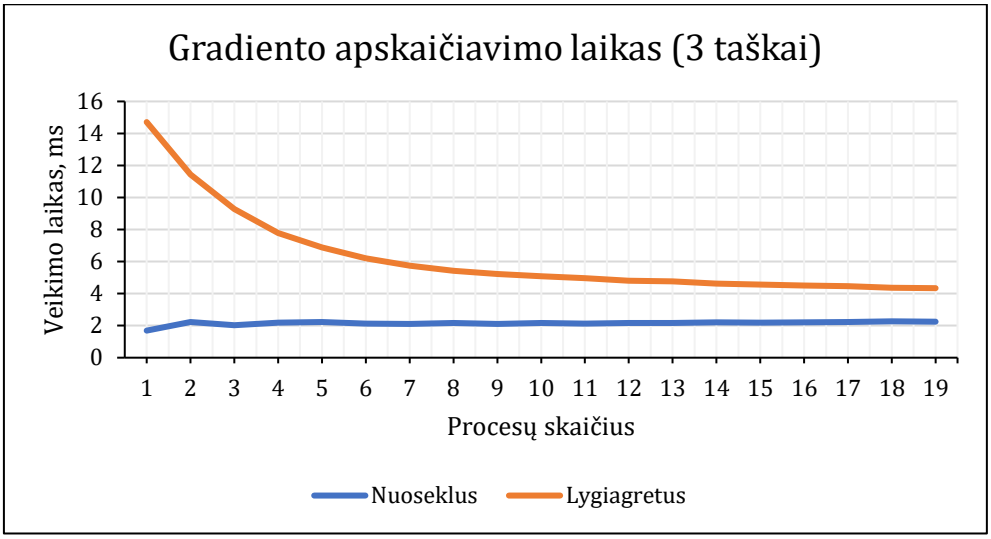
Lygiagretumo tyrimas

Tyrimą atlikau duotam parenkamų taškų kiekiui keičiant procesų skaičių nuo 1 iki 19. Parenkamų taškų skaičių kiekiai: 3, 6, 9, 12, 15. Su didesniu duomenų skaičiumi tyrimo trukmė viršija 1 minutę, tad ties 15 taškų nutraukiau tyrimą.

Žemiau pateiktose diagramose gali pasirodyti, jog nuoseklaus programos varianto veikimo trukmė irgi priklauso nuo procesų skaičiaus pagal ašis, tačiau tai netiesa. Taip pavaizdavau duomenis, kad būtų galima lengvai patikrinti skirtumą tarp veikimo greičio naudojant lygiagretų programavimą ir nenaudojant.

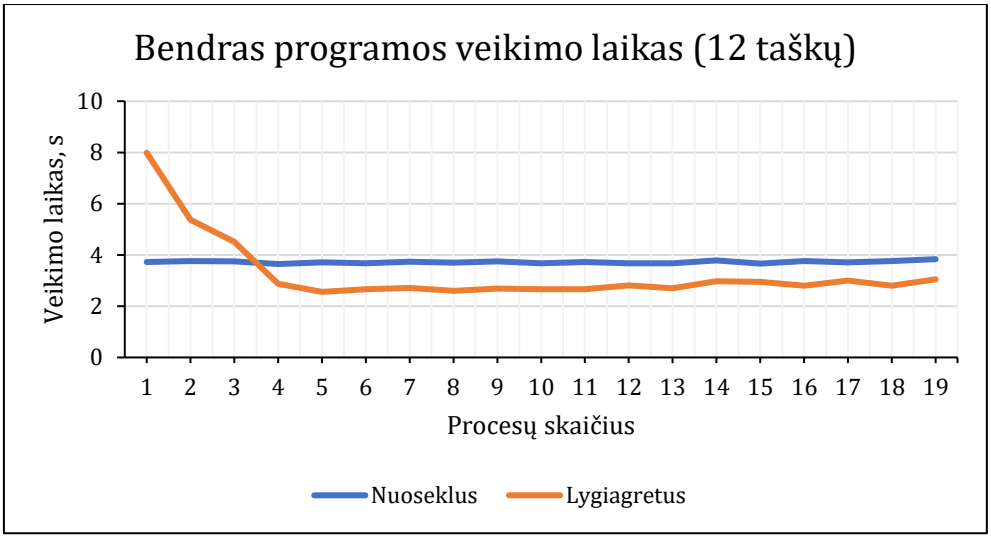


1 pav. Bendras programos veikimo laikas su 3 pasirinktais taškais

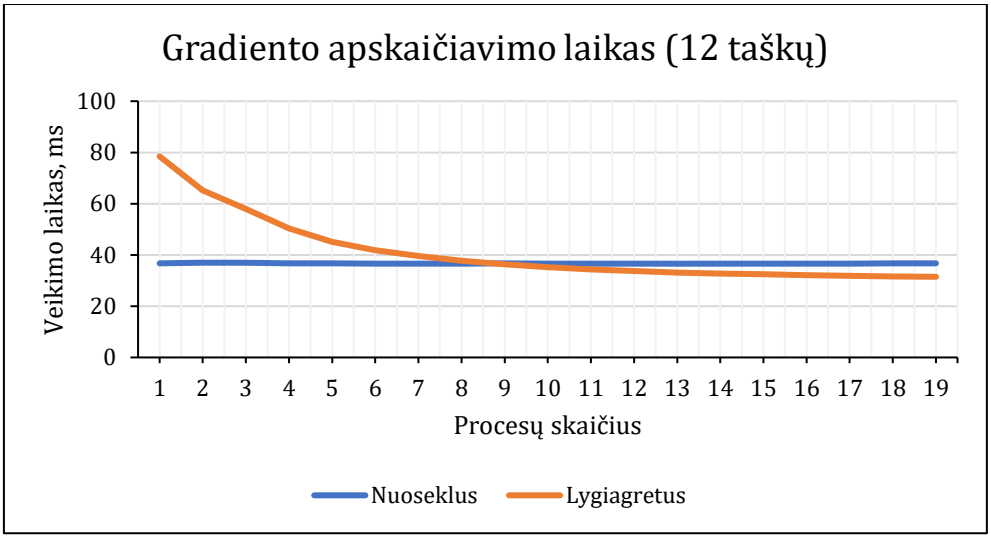


2 pav. Vidutinis gradiento apskaičiavimo laikas su 3 pasirinktais taškais

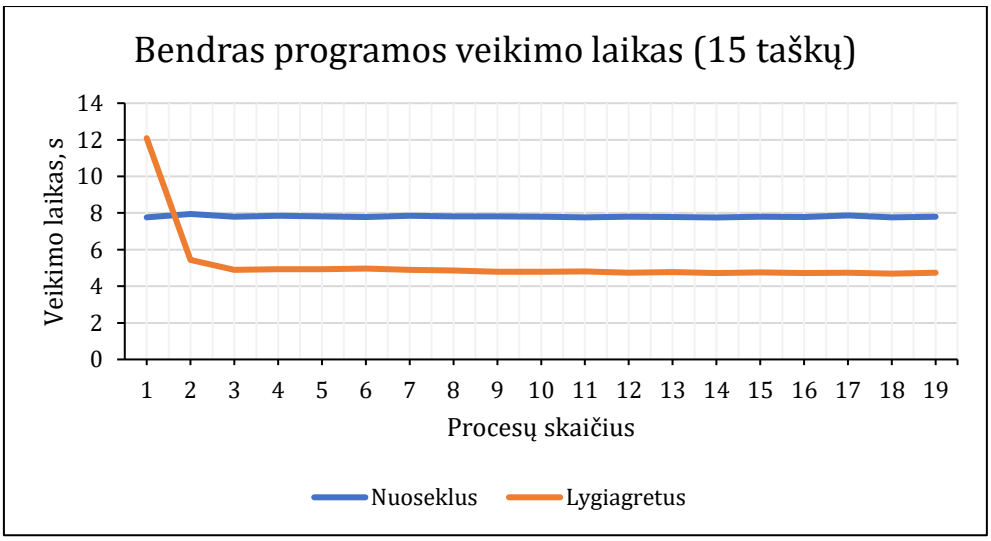
1 ir 2 paveikslėlyje pateiktų diagramų matosi, jog esant labai mažam duomenų skaičiui neapsimoka lygiagretinti programos, nes naujų procesų sukūrimo kaina laiko prasme stipriai viršija skaičiavimo laiko trukmę. Tas pats galioja tiek su 6 pasirinktais taškais, tiek su 9 taškais. Dėl lygiagretumo gaunamas pagreitėjimas gaunamas su 12 ir daugiau pasirinktų taškų. Tai pavaizduota diagramose 3 – 6.



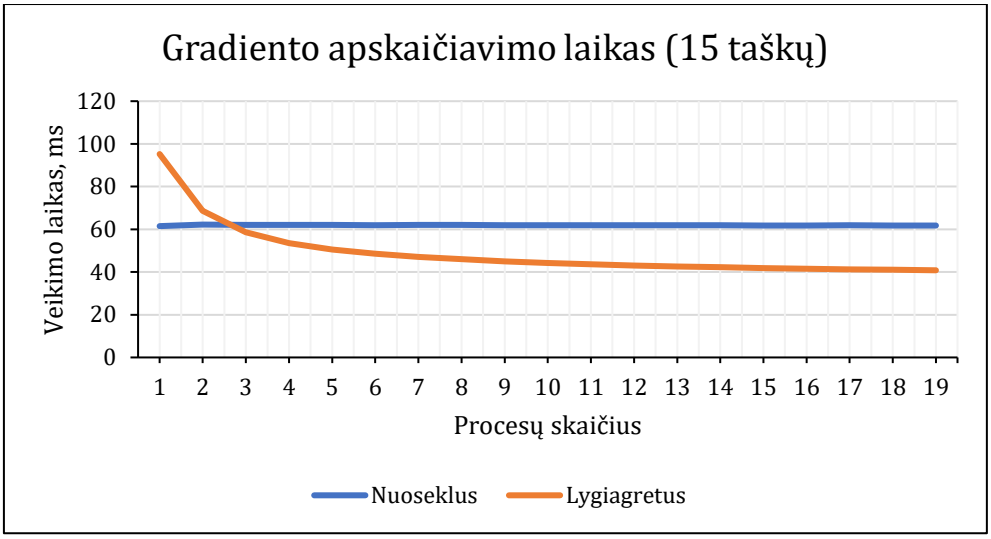
3 pav. Bendras programos veikimo laikas su 12 pasirinktų taškų



4 pav. Vidutinis gradiento apskaičiavimo laikas su 12 pasirinktų taškų



5 pav. Bendras programos veikimo laikas su 15 pasirinktų taškų



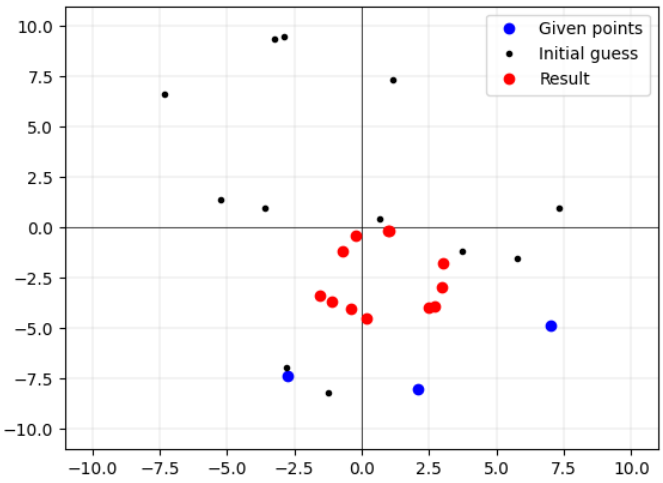
6 pav. Vidutinis gradiento apskaičiavimo laikas su 15 pasirinktų taškų

Kiekvienam duomenų atvejui pateikiau tiek bendrą programos vykdymo trukmę, tiek gradiento apskaičiavimo trukmę kad būtų galima pamatyti sunaudoto laiko priklausomybę nuo vienos kodo vietos išlygiagretinimo ir viso programos veikimo laiko. Bendras programos veikimo laikas įskaičiuoja ir procesų užimamą laiką, o gradiento apskaičiavimo trukmė parodo, kiek vidutiniškai per visas iteracijas buvo sugaišta laiko tik skaičiavimui, nekreipiant dėmesio į procesų valdymą. Dėl šios priežasties ir atsiranda skirtumas tarp kada išlygiagretinto kodo greitis aplenkia nuoseklaus kodo vykdymo greitį.

Rezultato korektiškumo patikrinimas

Įsitikinti, ar gautas rezultatas optimaliausias gana sunku. To nepavyks patikrinti skaičiuojant ranka, dėl labai didelio skaičiavimų kiekio. Neįmanoma pasakyti, ar gautas rezultatas yra optimaliausias, nes rastas minimumas priklauso nuo pradinio artinio ir algoritmo žingsnio dydžio. Taip pat programos eigoje gali atsitikti variantas, kai patenkama į lokalų minimumą.

Programos veikimo korektiškumą galima patikrinti atvaizdavus rezultatus grafiškai ir paanalizavus, ar jie atitinka sąlygą.



7 pav. Rezultatas su 12 pasirenkamų taškų

5 pav. pateiktas programos rezultatas, kai parenkama 12 taškų. Pagal sąlygą, jie turėtų priartėti prie duotų taškų ir jų atstumas turėtų būti artimas duotai reikšmei *S* (šiuo atveju 1). Tai ir matoma paveiksliuke.

Išvados

Iš gautų veikimo laikų, pavaizduotų 1 – 6 diagramose matyti, jog programos išlygiagretinimas ne visada pagreitins jos darbą. Esant mažam apdorojamų duomenų skaičiui naujų procesų sukūrimas užtrunka žymiai ilgiau nei pats rezultato apskaičiavimas.

Priklausomybės tarp procesų skaičiaus ir sugaišto laiko skirtumas bendroje kodo veikimo trukmėje ir konkrečiai gradiento apskaičiavime atsiranda dėl to, kad pirmuoju atveju įskaičiuojamas ir procesų valdymui skiriamas laikas. Veikimo trukmės nusistovėjimą didėjant procesų skaičiui galima paaiškinti ta pačia logika – kuo daugiau procesų, tuo daugiau laiko skiriama jiems valdyti.

Programinis kodas

gradient.py

```
import multiprocessing as mp
import time
import numpy as np
import utils
from calculations import M, N, psi, psi_prime_point

PROCESS_COUNT = 20
calc_times_mp = []
calc_times = []


def gradient_mp(stat_arr, new_arr, pool):
    pool_results = []

    tic = time.perf_counter()
    for i in range(M):
        pool_results.append(pool.apply_async(psi_prime_point, args=(stat_arr, new_arr, i)))

    g = [res.get() for res in pool_results]

    toc = time.perf_counter()
    calc_times_mp.append(toc - tic)

    return np.array(g) / np.linalg.norm(g)


def gradient(stat_arr, new_arr, _):
    g = []

    tic = time.perf_counter()
    for i in range(M):
        g.append(psi_prime_point(stat_arr, new_arr, i))
    toc = time.perf_counter()

    calc_times.append(toc - tic)

    return np.array(g) / np.linalg.norm(g)


def solve(gradient_function):
    pool = mp.Pool(processes=PROCESS_COUNT)
    stat_points = utils.read_points('data/points1.txt', N)
    new_points = utils.read_points('data/points2.txt', M)

    tic = time.perf_counter()

    initial_points = new_points
    step = 0.1
    iteration_count = 0
    old_psi = psi(stat_points, new_points)

    while step > 1e-3:
        iteration_count += 1

        old_points = new_points
        new_points = new_points - step * gradient_function(stat_points, new_points, pool)

        if utils.is_beyond_10(new_points):
            print("New points beyond 10")
            print(iteration_count, "iterations")
            return stat_points, initial_points, old_points

        new_psi = psi(stat_points, new_points)

        if new_psi > old_psi:
            step /= 5
            new_points = old_points
        else:
            step += step * 0.05

        old_psi = new_psi
```

```
    toc = time.perf_counter()
    pool.close()
    pool.join()

    # print(iteration_count, "iterations")
    return stat_points, initial_points, new_points, toc - tic


if __name__ == '__main__':
    coords = []

    for pc in range(1, 20):
        PROCESS_COUNT = pc

        given_points, initial_guess, result, total_time = solve(gradient)
        _, _, result_mp, total_time_mp = solve(gradient_mp)
        coords = [given_points, initial_guess, result]

        MULTIPLIER = 1000  # ms
        total_time *= MULTIPLIER
        total_time_mp *= MULTIPLIER
        gradient_avg = np.average(calc_times) * MULTIPLIER
        gradient_avg_mp = np.average(calc_times_mp) * MULTIPLIER

        print("{}, {:>10.2f}, {:>10.2f}, {:>10.2f}, {:>10.2f}".format(
            PROCESS_COUNT, total_time, total_time_mp, gradient_avg, gradient_avg_mp
        ))

        # print("Initial guess")
        # print(initial_guess)
        # print("Result (mp)")
        # print(result)
        # print("Result (non-mp)")

        # print("--- Non-MP Results ---")
        # print("{:>10.2f} s (total time)".format(total_time))
        # print("{:>10.2f} ms (avg gradient time)".format(gradient_avg))
        # print("--- Multiprocessing results ({} processes) ---".format(PROCESS_COUNT))
        # print("{:>10.2f} s (total time)".format(total_time_mp))
        # print("{:>10.2f} ms (avg gradient time)".format(gradient_avg_mp))
        # print(result_mp)

    utils.plot_results(coords[0], coords[1], coords[2])
```

calculations.py

```
from sys import exit
import numpy as np

N = 3   # Given points
M = 15  # Chosen points
S = 1


# Average distance
def avg_dist(stat_arr, new_arr):
    total = 0
    amount = 0

    for point in np.concatenate((stat_arr, new_arr)):
        total += np.sqrt(point[0] ** 2 + point[1] ** 2)
        amount += 1

    return total / amount


# Distance between stationary and new points
def v1(stat_arr, new_arr, d):
    total = 0

    for new_point in new_arr:
        for stat_point in stat_arr:
            total += abs(np.sqrt((new_point[0] - stat_point[0]) ** 2 + (new_point[1] -
stat_point[1]) ** 2) - d)

    return total


# Distance between new points
def v2(new_arr, d):
    total = 0

    for i in range(M):
        for j in range(i + 1):
            total += abs(np.sqrt((new_arr[i][0] - new_arr[j][0]) ** 2 + (new_arr[i][1] -
new_arr[j][1]) ** 2) - d)

    return total


# Distance from center
def v3(new_arr):
    total = 0

    for point in new_arr:
        total += abs(np.sqrt(point[0] ** 2 + point[1] ** 2) - S)

    return total


def psi(stat_arr, new_arr):
    d = avg_dist(stat_arr, new_arr)

    return v1(stat_arr, new_arr, d) + v2(new_arr, d) + v3(new_arr)


def psi_prime(stat_arr, new_arr, i, j):
    h = 0.001
    increased_arr = np.copy(new_arr)
    increased_arr[i][j] += h

    psi1 = psi(stat_arr, new_arr)
    psi2 = psi(stat_arr, increased_arr)

    if psi2 - psi1 == 0:
        print("Derivative is 0")
        exit(1)

    return (psi2 - psi1) / h
```

```
def psi_prime_point(stat_arr, new_arr, index):
    return [
        psi_prime(stat_arr, new_arr, index, 0),
        psi_prime(stat_arr, new_arr, index, 1)
    ]
```

utils.py

```
import matplotlib.pyplot as plt
import numpy as np


def is_beyond_10(points):
    for i in points:
        if abs(i[0]) > 10 or abs(i[1]) > 10:
            return True

    return False


def read_points(filename, amount):
    file = open(filename, 'r')
    points = []

    for i in range(amount):
        coords = file.readline().split(' ')
        points.append([
            float(coords[0]),
            float(coords[1]),
        ])

    return np.array(points)


def plot_results(given_points, initial_guess, result):
    plt.xlim(-11, 11)
    plt.ylim(-11, 11)
    plt.axvline(linewidth=0.5, color='k')
    plt.axhline(linewidth=0.5, color='k')
    plt.grid(linewidth=0.2)

    given_point = None
    initial_point = None
    result_point = None

    for p in given_points:
        given_point, = plt.plot(p[0], p[1], 'bo')

    for p in initial_guess:
        initial_point, = plt.plot(p[0], p[1], 'ko', markersize=3)

    for p in result:
        result_point, = plt.plot(p[0], p[1], 'ro')

    plt.legend(
        [given_point, initial_point, result_point],
        ["Given points", "Initial guess", "Result"]
    )

    plt.show()
```